Radboud Repository

Radboud University Nijmegen

PDF hosted at the Radboud Repository of the Radboud University Nijmegen

The version of the following full text has not yet been defined or was untraceable and may differ from the publisher's version.

For additional information about this publication click this link.
http://hdl.handle.net/2066/18934

Please be advised that this information was generated on 2017-12-05 and may be subject to change.

Formal Specification and Verification of JavaCard's Application Identifier Class

J.A.G.M. van den Berg, B.P.F. Jacobs, E. Poll

Computing Science Institute/

CSI-R0014 September 2000

Computing Science Institute Nijmegen
Faculty of Mathematics and Informatics
Catholic University of Nijmegen
Toernooiveld 1
6525 ED Nijmegen
The Netherlands

Formal Specification and Verification of JavaCard's Application Identifier Class*

Joachim van den Berg, Bart Jacobs, Erik Poll

Dept. Computer Science, Univ. Nijmegen,
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands.
{joachim, bart, erikpoll}@cs.kun.nl
http://www.cs.kun.nl/~{joachim, bart, erikpoll}

Abstract This note discusses a verification in PVS of the `AID` (Application Identifier) class from JavaCard's API. The properties that are verified are formulated in the interface specification language JML. This language is also used to express the properties that are assumed about the native methods from the `Util` class that are used in the `AID` class.
Keywords: JavaCard, JML, program verification, specification.
Classification: 68Q55, 68Q60, 68Q65 (MSC 2000);
D.1.5, D.2.4, F.3.1, F.3.2, F.4.1 (CR'98).

1 Introduction

This short note describes the obvious next step after the formal specification in JML [9,8] of JavaCard's API in [11], namely actual verification that the current reference implementation (version 2.1) [1] satisfies these specifications. This verification is done with the proof tool PVS [10], based on the translation of Java and JML into PVS, as incorporated in the LOOP tool, see [12]. This verification forms a test, both for the JML specifications, and for the Java implementations (and of course also for the LOOP tool). Here we concentrate on a small part of the API, involving essentially only the classes `Util` and `AID`. The emphasis is not so much on the actual statements that are being verified (because they are not so spectacular), but on how the results and assumptions are formulated and used in proofs. An earlier case study [6] in library class verification concentrated on the `Vector` class. It did not use a formal semantics of JML.

The JavaCard API specifications in JML make many implicit assumptions explicit, and provide useful (and hopefully readable) documentation, see [11]. But one can also make mistakes in writing specifications, so it

* To appear in the Proceedings of the JavaCard Workshop, Cannes, Sept. 2000.

is important to actually verify them, if possible. One option is static checking, like with ESC/Java [13]. This is certainly very effective and useful, but only allows (automatic) checking of certain "simple" properties, without guarantee of absolute correctness[1]. We see our approach as complementary, for providing additional certainty: our verifications are based on a mathematical semantics of Java (expressed in the logic of PVS) that allows in principle arbitrarily complex assertions. But the activity of proving such assertions is interactive, and far from automatic. The fact that both static and semantic analysis (and even run-time checking of assertions [3]) is possible for JML is a strong advantage of this specification language.

This paper can only give an impression of some of the issues in API verification. It discusses these issues via an example.

2 JavaCard's `Util` class: specification

The `AID` class makes use of two methods from the `Util` class, namely `arrayCopy` and `arrayCompare`. Both these methods are static, final and native: 'static' means that they can be invoked without a receiving object, 'final' that they cannot be overridden, and 'native' that no actual Java code is provided, but their implementation is given in some other (low-level) language.

The behaviour of these methods is described in informal comments in the `Util.java` file. We quote these explanations:

`arrayCopy`: "Copies an array from the specified source array, beginning at the specified position, to the specified position of the destination array (non-atomically)."

`arrayCompare`: "Compares an array from the specified source array, beginning at the specified position, with the specified position of the destination array from left to right. Returns the ternary result of the comparison : less than(-1), equal(0) or greater than(1)."

More information is given in the javadoc clarifications, see [1], especially about when exceptions may be expected. Such information is important for correct use of the JavaCard API in applets.

[1] It may be interesting to note that the verification of the `getBytes` method that is discussed in Section 3 below has also been done in ESC/Java, but without the modifies clauses. The ESC/Java tool accepted this example in only a few seconds, indicating that it saw no errors. (With thanks to Jim Saxe from Compaq SRC for demonstrating this example.)

```
public static final native short arrayCopy(byte[] src,
                                           short srcOff,
                                           byte[] dest,
                                           short destOff,
                                           short length)
throws ArrayIndexOutOfBoundsException,
       NullPointerException,
       TransactionException;

/*@ behavior
  @    requires: src != null && srcOff >= 0 &&
  @                              srcOff + length <= src.length &&
  @              dest != null && destOff >= 0 &&
  @                               destOff + length <= dest.length &&
  @              length >= 0;
  @  modifiable: dest[*];
  @     ensures: true;
  @     signals: (TransactionException) true;
  @*/

public static final native byte arrayCompare(byte[] src,
                                             short srcOff,
                                             byte[] dest,
                                             short destOff,
                                             short length)
throws ArrayIndexOutOfBoundsException,
       NullPointerException;

/*@ normal_behavior
  @    requires: src != null && srcOff >= 0 &&
  @                              srcOff + length <= src.length  &&
  @              dest != null && destOff >= 0 &&
  @                               destOff + length <= dest.length &&
  @             length >= 0;
  @  modifiable: \nothing;
  @     ensures: true;
  @*/
```

Figure1. JML specifications for `arrayCopy` and `arrayCompare` from `Util`

We shall use formal specifications in JML, as described in Figure 1 (see also [11]). They concentrate on normal/abrupt termination and modification, and say nothing about the functional behaviour of these methods. Hence our specifications are clearly incomplete (at this stage). But they do already convey useful information (and are strong enough to prove the `AID` specifications in the next section).

Instead of explaining JML in general, we explain only the meaning of these specifications. The `behavior` keyword indicates that if the precondition as given by the `requires` clause holds, then either the method terminates normally, and the "normal" postcondition after the `ensures` keyword holds, or the method terminates abruptly because of an exception of the type indicated after `signals`, and the ensuing "abrupt" postcondition holds. The keyword `normal_behavior`, instead of just `behavior`, indicates that the method must terminate normally, provided the precondition holds. The `behavior` specifications are translated into partial Hoare triples, and the `normal_behavior` specifications into total Hoare triples, in a special version of Hoare logic adapted to Java [5]. The `modifiable` clauses tell which items may be changed by the method, and thus also, implicitly, which are left unaltered. The method `arrayCompare` does not modify anything, and thus has no side-effect. The `arrayCopy` method can modify the entries of its parameter array `dest`, as indicated by `dest[*]`. Actually, one can be more precise and say that this method only modifies the entries of `dest` in the range `offset`, ..., `offset + length - 1`, but that is not needed at this stage. The `TransactionException` may occur when an overflow arises in JavaCard's transaction buffer—which is used to enable rollback of operations in case of failure.

JML is expressive enough to formulate also functional behaviour. For instance, for `arrayCopy` we could have used as `ensures` clause:

```
dest == \old(dest) &&
dest.length == \old(dest.length) &&
\forall(int i) [ i >= srcOff && i <= srcOff + length - 1
                   ==> dest[i] == \old(src[i]) ] &&
\forall(int i) [ (i >= 0 && i < srcOff) ||
                 (i < dest.length && i > srcOff + length - 1)
                   ==> dest[i] == \old(dest[i]) ] &&
\result == dstOff + length
```

But this property is not needed for the verification below.

3 JavaCard's `AID` class: specification and verification

In the JavaCard platform each applet instance and package is uniquely identified and selected by an application identifier (AID)—and not, as

usual, by a string possibly in combination with a domain name, see [4, §§3.8]. AIDs are used in loading and linking. As prescribed in the ISO 7816 standard, each AID consists of an array of bytes, ranging in length from 5 to 16. The first five bytes form what is called the resource identifier (RID), which is assigned by ISO to a company. The possible remainder (between 0 and 11 bytes) forms the proprietary identifier extension (PIX), which is under the control of individual companies.

Space restrictions prevent us from discussing the `AID` class in full, so we concentrate on the essentials. Its field, constructor and five methods, without their implementations, look as follows.

```
public final class AID{
  byte[] theAID;
  public AID(byte[] bArray, short offset, byte length) ...
  public byte getBytes(byte[] dest, short offset) ...
  public boolean equals(Object anObject) ...
  public boolean equals(byte[] bArray, short offset, byte length) ...
  public boolean partialEquals(byte[] bArray, short offset, byte length) ...
  public boolean RIDEquals (AID otherAID) ...
}
```

Our JML specification of this class adds a class invariant, and pre-/post-conditions for its constructor and methods. Since the array of an AID consists of a 5-byte RID possibly together with a PIX of up-to 11 bytes, our invariant is (as also mentioned in [11]):

```
//@ invariant : theAID != null &&
                5 <= theAID.length && theAID.length <= 16;
```

The proof obligation is to show that this property holds after termination (both normal and abrupt) of the constructor, and also that it holds after termination (both normal and abrupt) of each method, assuming it holds in the state in which the method is invoked[2].

Here we shall concentrate on the method `getBytes`. Its official explanation is: "Called to get the AID bytes encapsulated within AID object." More precisely, after a successful method invocation `getBytes(dest, offset)`, the contents of the `theAID` byte array can be found at the parameter entries `dest[offset]`, ..., `dest[offset + theAID.length - 1]`. The JML specification of `getBytes` is:

```
/*@ behavior
  @    requires: dest != null && offset >= 0 &&
  @              offset + theAID.length <= dest.length
  @  modifiable: dest[*];
```

[2] What is surprising is that the `theAID` field is not declared as `private`. As it stands, it can be modified from the outside (but only within its package), making it vulnerable to a breach of the invariant. We consider the ommission of the `private` access modifier a bug.

```
@     ensures: true;
@     signals: (TransactionException) true;
@*/
```

Notice that it only mentions termination and modification behaviour. The implementation of `getBytes` in the JavaCard API version 2.1 consists of only two lines:

```
{
  Util.arrayCopy(theAID, (short)0, dest, offset, (short)theAID.length);
  return (byte) theAID.length;
}
```

We briefly discuss the proof in PVS that this implementation satisfies the specification, assuming the `arrayCopy` method from `Util` satisfies its specification in Figure 1. The main distinction in the proof is between normal and abrupt termination of the `getBytes` method.

Normal termination If the precondition and the invariant hold and `getBytes` terminates normally, then we have to show two properties about the post-state, namely:

1. We must show that only the contents of the `dest` array have changed, and nothing else. In the memory model (see [2]) in the semantics of Java underlying the LOOP translation, the `dest` parameter refers to a particular memory cell. What we prove—the LOOP translation of the clause `modifiable dest[*]`—is that all memory cells except this one remain unaltered[3]. The modification property of `getBytes` follows easily from the one for `arrayCopy`, see Figure 1. In order to be able to use this `arrayCopy` specification, its `requires` clause (plus invariant, if any) has to be established. But that's easy in this case, using the `getBytes requires` clause and the `AID` invariant.
2. We must show that the `AID` invariant holds in the post-state. This is easy by the assumption that the invariant holds in the pre-state and the fact that `theAID` is not modified by `getBytes`. Even if `theAID` and `dest` are aliases[4] the invariant is maintained, because the array reference `dest` itself and the length of the array referred to by `dest` are not modified: the modifies clause only says that the array components `dest[*]` may be modified.

[3] Since this translation of the modifies clause is expressed in term of cells in the underlying memory model, it is not restricted to the currently known fields (stored in already known cell positions), but it can also be used for currently unknown fields that are introduced in subclasses in the future.

[4] This aliasing possibility is easily overlooked, but our semantics forces us to consider it explicitly.

Abnormal termination If the precondition and the invariant hold and `getBytes` terminates abruptly because of an exception, then we have to show that this exception is an instance of `TransactionException`, and that the "abnormal" post-state still satisfies the two properties:

1. Only the contents of `dest` have changed;
2. The `AID` invariant still holds.

The proof is much like in the normal case, and relies on the specification of `arrayCopy`. During the proof we establish explicitly that the (assumed) abrupt termination of `getBytes` is due to abrupt termination of `arrayCopy`.

Specifications similar to the specification of `getBytes` have been written for the other `AID` methods and constructor. They have all been proved in PVS, assuming the `Util` specifications in Figure 1. In the end, what we have reached is a fully verified formal specification of a small part of the JavaCard API. It provides a reliable basis for building JavaCard applets. In due time, these specifications will be published on the web [12].

One interesting point in the semantics of specifications came up during the verification, namely about the validity of class invariants. It is tempting to use some sort of global assumption that every object which is an instance of a class `A` satisfies the invariant of `A`. But this easily leads to inconsistencies, for example because invariants should hold at the beginning and end of method bodies, but may be broken (temporarily) in between. What we have used instead is that the LOOP tool strengthens all pre-conditions with an assertion that a parameter (if any) of some class `A` satisfies the `A`-invariant. For example, the pre-condition of `RIDEquals` is strengthened to include an assertion that the parameter `otherAID` of type `AID` satisfies the `AID`-invariant. This may then be used during verification of the method. But when the method specification is used, this pre-condition has to be established, which includes actual verification of the invariants of the classes of the parameters. However, this does not cover all cases. For example, if we have an object `obj` of class `Object`, and (down)cast it to `A`, then we would like to use that `(A)obj` satisfies the `A`-invariant[5]. How to handle this in general still has to be elaborated. More on invariants in an object-oriented setting can be found in [7].

It is not so easy to quantify the effort that was needed for the whole `AID` verification, because we have used this case study as an experiment

[5] This comes up during verification of the `equals` method from `AID` where we have solved this problem by adding an extra assumption in the `requires` clause.

to try out various semantical descriptions and proof methods. But, to give a rough impression, handling all the possible cases for the `getBytes` method in PVS takes about one hour of user interaction.

4 Conclusions

We have sketched the `AID` verification in PVS of its JML specification, based on the semantics provided by the LOOP tool. The verification involves dealing explicitly with many possible cases, some of which may be easily overlooked. This is of course a very modest exercise, but it does show the feasibility of such validation efforts for class libraries. They will form the basis for actual applet verification at a later stage. Initial goals for these applet verifications will be proving the absence of runtime exceptions and non-termination, and proving the absence of unwanted side effects (notably side effects on other applets) expressed by `modifiable` clauses.

References

1. JavaCard API 2.1. `http://java.sun.com/products/javacard/htmldoc/`.
2. J. van den Berg, M. Huisman, B. Jacobs, and E. Poll. A type-theoretic memory model for verification of sequential Java programs. In D. Bert and C. Choppy, editors, *Recent Trends in Algebraic Development Techniques*, number 1827 in Lect. Notes Comp. Sci. Springer, Berlin, 2000.
3. A. Bhorkar. A run-time assertion checker for Java using JML. Techn. Rep. 00-08, Dep. of Comp. Sci., Iowa State Univ. (`http://www.cs.iastate.edu/~leavens/JML.html`), 2000.
4. Z. Chen. *Java Card Technology for Smart Cards.* The Java Series. Addison-Wesley, 2000.
5. M. Huisman and B. Jacobs. Java program verification via a Hoare logic with abrupt termination. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, number 1783 in Lect. Notes Comp. Sci., pages 284–303. Springer, Berlin, 2000.
6. M. Huisman, B. Jacobs, and J. van den Berg. A case study in class library verification: Java's Vector class. Techn. Rep. CSI-R0007, Comput. Sci. Inst., Univ. of Nijmegen. Conditionally accepted for publication in *Software Tools for Technology Transfer*, 2000.
7. K. Huizing, R. Kuiper, and SOOP. Verification of object oriented programs using class invariants. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, number 1783 in Lect. Notes Comp. Sci., pages 208–221. Springer, Berlin, 2000.
8. G.T. Leavens, A.L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov and B. Rumpe, editors, *Behavioral Specifications of Business and Systems*, pages 175–188. Kluwer, 1999.

9. G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Techn. Rep. 98-06, Dep. of Comp. Sci., Iowa State Univ. (`http://www.cs.iastate.edu/~leavens/JML.html`), 1999.
10. S. Owre, J.M. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Trans. on Softw. Eng.*, 21(2):107–125, 1995.
11. E. Poll, J. van den Berg, and B. Jacobs. Specification of the JavaCard API in JML. In *Fourth Smart Card Research and Advanced Application Conference (CARDIS).* Kluwer Acad. Publ., 2000, to appear. Available as Techn. Rep. CSI-R0005, Comput. Sci. Inst., Univ. of Nijmegen.
12. Loop Project. `http://www.cs.kun.nl/~bart/LOOP/`.
13. Extended static checker ESC/Java. Compaq System Reserch Center. `http://www.research.digital.com/SRC/esc/Esc.html`.